Hibernate

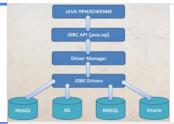
Что такое ORM? Что такое JPA? Что такое Hibernate?

ORM(Object Relational Mapping) - это концепция преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот.
JPA(Java Persistence API) - это стандартная для Java спецификация, описывающая принципы ORM. JPA не умеет работать с объектами, а только определяет правила как должен действовать каждый провайдер (Hibernate, EclipseLink), реализующий стандарт JPA
(Plain Old Java Object - POJO)
Hibernate - библиотека, являющаяся реализацией этой спецификации, в которой можно использовать стандартные API-интерфейсы JPA.

JDBC - Java DataBase Connectivity — API для работы с реляционными (зависимыми) БД. Платформенно независимый промышленный стандарт взаимодействия Java приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE. Предоставляет методы для получения и обновления данных. Не зависит от конкретного типа базы. Библиотека, которая входит в стандартную библиотеу, содержит: набор классов и интерфейсов для работы с БД (для нас разработчиков api) + интерфейсы баз данных.

Наш код –> JDBC –> драйвера разработчиков БД –> БД.
JDBC has 3 entities:
1. Connection (класс). Объект которого отвечает за соединение с базой и режим работы с ней (лекция 3.1.4)
2. Statement (объект для оператора JDBC) используется для отправки SQL-оператора на сервер баз данных. Объект для оператора связан с объектом Connection и является объектом, обрабатывающим взаимодействие между приложением и сервером баз данных. Можно:
- что-то поменять Update statement (create, delete, insert) в базе;
- что-то запросить Query statement (select) из базы;
Для разных задач есть разные виды Statement-ов:
- statement – обычный. передаем в него либо Update либо Query
- PreparedStatement - возможность сделать некий шаблон запроса, подставлять в него к-то значения и использовать его
- CallableStatement - предоставляет возможность вызова хранимой процедуры, расположенной на сервере, из Java™-приложения.
3. ResultSet. Объект с результатом запроса, который вернула база. Внутри него таблица.

Важные интерфейсы Hibernate:
Session - обеспечивает физическое соединение между приложением и БД. Основная функция - предлагать DML-операции для экземпляров сущностей.
SessionFactory - это фабрика для объектов Session. Обычно создается во время запуска приложения и сохраняется для последующего использования. Является потокобезопасным объектом и используется всеми потоками приложения.
Transaction - однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC транзакций. Session может занимать несколько Transaction в определенных случаях, является необязательным API.
Query - интерфейс позволяет выполнять запросы к БД. Запросы написаны на HQL или на SQL.



Что такое EntityManager?

EntityManager интерфейс JPA, используемый для взаимодействия с персистентным контекстом, который описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA.
Основные операции:
1) **Операции над Entity**: persist (добавление Entity), merge (обновление), remove (удаления), refresh (обновление данных), detach (удаление из управление JPA), lock (блокирование Entity от изменений в других thread),
2) **Получение данных**: find (поиск и получение Entity), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery
3) **Получение других сущностей JPA**: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate
4) Работа с EntityGraph: createEntityGraph, getEntityGraph
5) Общие операции над EntityManager или всеми Entities: close, clear, isOpen, getProperties, setProperty.

Персистентный контекст - это набор экземпляров сущностей, загруженных из БД или только что созданных. Персистентный контекст является своего рода **кэшем данных в рамках транзакции - это и есть кэш первого уровня.** Внутри контекста персистентности происходит управление экземплярами сущностей и их жизненным циклом. EntityManager автоматически сохраняет в БД все изменения, сделанные в его персистентном контексте, в момент коммита транзакции, либо при явном вызове метода flush().

JPA	JDBC по аналогии	Hibernate
EntityManagerFactory	DataSource	SessionFactory
EntityManager	Connection	Session
JPQL		HQL

Объекты EntityManager не являются потокобезопасными. Это означает, что каждый поток должен получить свой экземпляр EntityManager, поработать с ним и закрыть его в конце.

Каким условиям должен удовлетворять класс чтобы являться Entity?

Entity это легковесный хранимый объект бизнес логики. Основная программная сущность это entity-класс, который так же может использовать дополнительные классы, которые могут использоваться как вспомогательные классы или для сохранения состояния entity.

Персистентность в программировании означает способность состояния существовать дольше, чем процесс, создавший его. Persistence context — это среда в которой экземпляры сущностей синхронизируются с аналогичными сущностями в базе данны

Entity (Сущность) — POJO-класс, связанный с БД (не финальный класс верхнего уровня помеченный аннотацией @Entity, содержащий первичный ключ, пустой public или protected конструктор, и доступ к полям, должен быть доступен для других классов только через getter/setter (или другим методам бизнес логики)

1) Entity класс должен быть помечен аннотацией Entity или описан в XML файле
2) Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами) - при получении данных из БД и формировании из них объекта сущности, Hibernate должен создать этот самый объект сущности,
3) Entity класс должен быть классом верхнего уровня (top-level class),
4) Entity класс не может быть enum или интерфейсом,
5) Entity класс не может быть финальным классом (final class),
6) Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables),
7) Если объект Entity класса будет передаваться по значению как отдельный объект (detached object), например через удаленный интерфейс (through a remote interface), он так же должен реализовывать Serializable интерфейс (чтобы объекты которые достаются из базы могли сохраняться в кэше).
8) Поля Entity класс должны быть напрямую доступны только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Такие классы должны обращаться только к методам (getter/setter методам или другим методам бизнес-логики в Entity классе),
9) Entity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов которые уникально определяют запись этого Entity класса в базе данных

Может ли абстрактный класс быть Entity?

Может, при этом он сохраняет все свойства Entity, отличается от обычных Entity классов только тем, что нельзя создать объект этого класса. Абстрактные Entity классы используются в наследовании, когда их потомки наследуют поля абстрактного класса

Может ли Entity класс наследоваться от не Entity классов (non-entity classes)?

Может

Может ли Entity класс наследоваться от других Entity классов?

Может

Может ли не Entity класс наследоваться от Entity класса?

Может

Что такое встраиваемый (Embeddable) класс? Какие требования JPA устанавливает

Embeddable класс - это класс, который не используется сам по себе, а является частью одного или нескольких Entity-классов. Entity-класс может содержать как одиночные встраиваемые классы, так и коллекции таких классов. Также такие классы могут быть использованы как ключи или значения map. **Во время выполнения каждый встраиваемый класс принадлежит только одному объекту Entity-класса и не может быть использован для передачи данных между объектами Entity-классов (то есть такой класс не является общей структурой данных для разных объектов). В целом, такой класс служит для того чтобы выносить определение общих атрибутов для нескольких Entity.**

1. Такие классы должны удовлетворять тем же правилам что Entity классы, за исключением того что они **не обязаны содержать первичный ключ и быть отмечены аннотацией Entity**
2. Embeddable класс должен быть помечен аннотацией @Embeddable или описан в XML файле конфигурации JPA. А поле этого класса в Entity аннотацией @Embedded. **От Embeddable класса нельзя наследоваться - конечный класс Стоит использовать если класс конечный**

То есть, если класс Person с полями name и age встроен и в класс Driver, и в класс Baker, то у обоих последних классов появятся оба поля из класса Person. Но если у объекта Driver эти поля будут иметь значения “Иван” и “35”, то эти же поля у объекта B

```
@Embeddable
public class Person {
    private String name;
    private int age;
}
```

Embeddable-класс может содержать другой встраиваемый класс. Embeddable Person может имент поле класса Adress - тогда в Student пишем adress.имяполя.

устанавливает к встраиваемым (Embeddable) классам?

Встраиваемый класс может содержать связи с другими Entity или коллекциями Entity, если такой класс не используется как первичный ключ или ключ map'ы.
Так как мы можем встраивать классы в неограниченное количество других классов, то у каждого класса, содержащего встраиваемый класс, мы можем изменить названия полей из встраиваемого класса.
Например, у класса Driver поля из встраиваемого класса Person будут изменены с name на driver_name и с age на driver_age. Делаем с помощью аннотации AttributeOverride - также класс Embeddable "Person" может быть встроет в Studen и с помощью AttributeOverride можно задать для 2 персон "муж и жен пол" 2 разделения. @AssociationOverride - в Embeddable Person есть объект Embeddable Adress а в нем связть MonyToOne с другой таблицой Street для этого в Studen ставим аннотацию @AssociationOverride над Person и подключаем @AssociationOverride(name = adress.street joinColumns = @JoinColumn(name))

```
@Entity
public class Driver {

    @Embedded
    @AttributeOverrides({
        @AttributeOverride( name = "name",
                            column = @Column(name = "driver_name")),
        @AttributeOverride( name = "age",
                            column = @Column(name = "driver_age"))
    })
    private Person person;
    ...
}
```

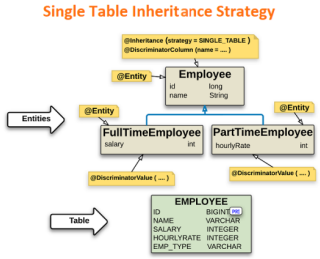
Что такое Mapped Superclass?

Mapped Superclass - (позволяет обойти блокировку наследования Embeddable-классов, то есть от такого класса мы можем наследоваться, также в классе Entity если мы используем класс MappedSuperclass то над ним нужно ставить анттотацию @Embedded) это класс, от которого наследуются Entity , он может содержать аннотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования установленные для Entity (например, он может не содержать первичного ключа). Такой класс не может использоваться в операциях EntityManager или Query. Такой класс должен быть отмечен аннотацией MappedSuperclass или описан в xml файле.
Создание такого класса-предка оправдано тем, что мы заранее определяем ряд свойств и методов, которые должны быть определены в сущностях. Использование такого подхода позволило сократить количество кода.

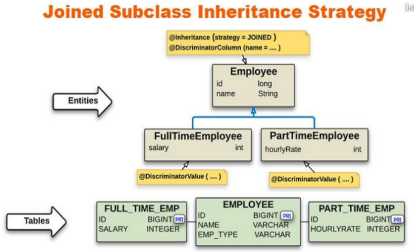
Совокупность полей и методов (свойств) сущности называется атрибутами.

Какие три типа стратегий наследования мапинга (Inheritance Mapping Strategies) описаны в JPA?

Inheritance Mapping Strategies **Стратегии наследования нужны для того, чтобы дать понять провайдеру (Hibernate) как ему отображать в БД сущности-наследники:**
1) Одна таблица на всю иерархию классов (SINGLE_TABLE) — все enity, со всеми наследниками записываются в одну таблицу, для идентификации типа entity определяется специальная колонка "discriminator column". Например, если есть entity Animals с классами-потомками Cats и Dogs, при такой стратегии все entity записываются в таблицу Animals, но при это имеют дополнительную колонку animalType в которую соответственно пишется значение «cat» или «dog». **Минусом является то что в общей таблице, будут созданы все поля уникальные для каждого из классов-потомков, которые будет пусты для всех других классов-потомков.** Например, в таблице animals окажется и скорость лазанья по дереву от cats и может ли пес приносить тапки от dogs, которые будут всегда иметь null для dog и cat соотвественно.
Минусом стратегии является невозможность применения ограничения NOT NULL для тех колонок таблицы, которые характерны только для классов-наследников., но можно использовать тригеры. Является стратегией по умолчанию. @DiscriminatorColumn(name = "EMP_TYPE") - имя общей колонки указывающий на принодлежность к классу @DiscriminatorValue("P") - указывает какое имя будет отображенно в @DiscriminatorColumn



2) Стратегия «соединения» (JOINED) — В данной стратегии корневой класс иерархии представлен отдельной таблицей, а каждый класс-наследник имеет свою таблицу, в которой отображены только поля этого класса-наследника., дополнительно устанавливается связь (relationships) между этими таблицами, например в случае классов Animals (см.выше), будут три таблицы animals, cats, dogs, причем в cats будет записана только ключ и скорость лазанья, в dogs — ключ и умеет ли пес приносить палку, а в animals все остальные данные cats и dogs с ссылкой на соответствующие таблицы. **Минусом тут являются потери производительности от объединения таблиц (join) для любых операций. @Id, который должен быть определен только в родительской таблице.**

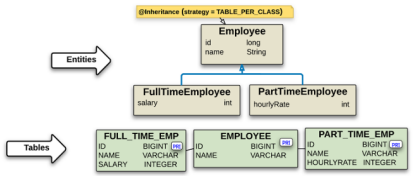



```
-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}
-- Native queries --
'Select * from Employee'
[F, 1, Sara]
[P, 2, Robert]
'Select * from FULL_TIME_EMP'
[100000, 1]
'Select * from PART_TIME_EMP'
[60, 2]
```

3) Таблица для каждого класса (TABLE_PER_CLASS) — каждый отдельный класс-наследник имеет свою таблицу.Во всех таблицах подклассов хранятся все поля этого класса плюс те, которые унаследованы от суперкласса. т.е. для cats и dogs (см.выше) все данные будут записываться просто в таблицы cats и dogs как если бы они вообще не имели общего суперкласса. **Минусом является плохая поддержка полиморфизма (polymorphic relationships) и то что для выборки всех классов иерархии потребуются большое количество отдельных sql запросов или использование UNION запроса.**

Для задания стратегии наследования используется аннотация Inheritance (или соответствующие блоки)
Кроме того, Hibernate поддерживает четвёртый, немного другой вид полиморфизма:

Неявный полиморфизм (implicit polymorphism)




```
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}
-- Native queries --
'Select * from Employee'
// no data
'Select * from FULL_TIME_EMP'
[1, Sara, 100000]
'Select * from PART_TIME_EMP'
[2, Robert, 60]
-- Loading entities --
List<Employee> entityAList = em.createQuery("Select t from Employee t")
.getResultList(); // Hibernate makes additional sql- or union-queries to get entities
PartTimeEmployee{id=2, name='Robert', hourlyRate='60'}
FullTimeEmployee{id=1, name='Sara', salary=100000}
```

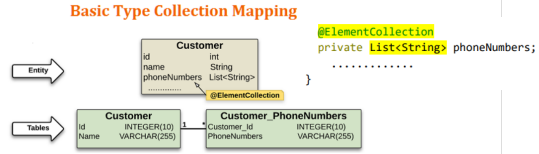
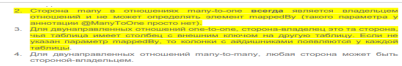
Как мапятся Enum'ы?

@Enumerated(EnumType.STRING) - означает, что в базе будут храниться имена Enum.
@Enumerated(EnumType.ORDINAL) - в базе будут храниться порядковые номера Enum. @Enumerated(EnumType.STRING) **Однако переименование значения enum все равно нарушит работу базы данных.** Кроме того, даже несмотря на то, что это представление данных гораздо более читаемо по сравнению с параметром @Enumerated(EnumType.ORDINAL), оно потребляет намного больше места, чем необходимо. Это может оказаться серьезной проблемой, когда нам нужно иметь дело с большим объемом данных.

Другой вариант - мы можем смапить наши enum в БД и обратно в методах с аннотациями **@PostLoad и @PrePersist. @EntityListener над классом Entity, в которой указать класс, в котором создать два метода, помеченнх этими аннотациями.**
Идея в том, чтобы в сущности иметь не только поле с Enum, но и вспомогательное поле. Поле с Enum аннотируем @Transient, а в БД будет храниться значение из вспомогательного поля. кажется неправильным иметь в сущности целых два атрибута, представляющих одно перечисление.

По порядковым номерам Если мы сохраняем в БД сущность, у которой есть поле-перечисление (Enum), то **в таблице этой сущности создаётся колонка для значений этого перечисления и по умолчанию в ячейки сохраняется порядковый номер этого перечисления (ordinal).**

	В JPA с версии 2.1 можно использовать Converter для конвертации Enum'а в некое его значение для сохранения в БД и получения из БД. Все, что нам нужно сделать, это создать новый класс, который реализует javax.persistence.AttributeConverter и аннотировать его с помощью **@Converter и поле в сущности аннотацией @Convert**. Мы установили @Converter(autoApply=true), чтобы JPA автоматически применял логику преобразования ко всем сопоставленным атрибутам типа Category. В противном случае нам пришлось бы поместить аннотацию @Converter непосредственно над полем Category у каждой сущности, где оно имеется. **Более того, мы можем безопасно добавлять новые значения enum или изменять существующие, не нарушая уже сохраненные данные.**	`@Converter(autoApply = true)`
		`public class CategoryConverter implements AttributeConverter<Category, String> {`
		`  @Override`
		`  public String convertToDatabaseColumn(Category category) {`

Как мапятся даты (до java 8 и после)?	Аннотация **@Temporal** до Java 8, **в которой надо было указать какой тип даты мы хотим использовать.** В Java 8 и далее аннотацию ставить не нужно. java.time Все классы в новом API неизменяемые (immutable) и, как следствие, потоко-безопасные. **Точность представления времени составляет одну наносекунду,** что в миллион раз точнее чем в пакете java.util.	Тип java.util.Date содержит информацию о дате и времени с точностью до миллисекунд.
Как "смапить" коллекцию примитивов?	**@ElementCollection** @OrderBy Если у нашей сущности есть поле с коллекцией, то мы привыкли ставить над ним аннотации @OneToMany либо @ManyToMany. Но данные аннотации применяются в случае, когда это коллекция других сущностей (entities). **Если у нашей сущности коллекция не других сущностей, а базовых или встраиваемых (embeddable) типов** для этих случаев в JPA имеется специальная аннотация **@ElementCollection, которая указывается в классе сущности над полем коллекции. Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем две таблицы: одну для сущности, вторую для коллекции элементов.** **При добавлении новой строки в коллекцию, она полностью очищается и заполняется заново, так как у элементов нет id. Можно решить с помощью @OrderColumn** @CollectionTable - позволяет редактировать таблицу с коллекцией, (CollectionTable отпределяет имя таблицы и @JoinColumn или @JoinColumns в случае составного первичного ключа.)	
Какие есть виды связей?	Существуют 4 типа связей: 1. OneToOne - когда один экземпляр Entity может быть связан не больше чем с одним экземпляром другого Entity. 2. OneToMany - когда один экземпляр Entity может быть связан с несколькими экземплярами других Entity. 3. ManyToOne - обратная связь для OneToMany. Несколько экземпляров Entity могут быть связаны с одним экземпляром другого Entity. 4. ManyToMany - экземпляры Entity могут быть связаны с несколькими экземплярами друг друга.	
	Каждую из которых можно разделить ещё на два вида: **1. Bidirectional** Владеемая сторона в двунаправленных отношениях должна ссылаться на владеющую сторону используя элемент mappedBy аннотаций @OneToOne, @OneToMany, или @ManyToMany. Элемент mappedBy определяет поле в объекте, который является владельцем отношения. **2. Undirectional** В однонаправленных отношениях только одна сущность имеет поле, которое ссылается на вторую сущность. Вторая сущность (сторона) не имеет поля первой сущности и не знает об отношениях. **Элемент mappedBy определяет поле в объекте, который является владельцем отношения.**	
	Bidirectional (Двунаправленные отношения) — ссылка на связь устанавливается у всех Entity, то есть в случае OneToOne A-B в Entity A есть ссылка на Entity B, в Entity B есть ссылка на Entity A. Entity A считается владельцем этой связи (это важно для случаев каскадного удаления данных, тогда при удалении A также будет удалено B, но не наоборот). **Undirectional**- ссылка на связь устанавливается только с одной стороны, то есть в случае OneToOne A-B только у Entity A будет ссылка на Entity B, у Entity B ссылки на A не будет.	
Что такое владелец связи?	В отношениях между двумя сущностями всегда есть одна владеющая сторона, а владеемой может и не быть, если это однонаправленные отношения. По сути, **у кого есть внешний ключ на другую сущность - тот и владелец связи.** То есть, если в таблице одной сущности есть колонка, содержащая внешние ключи от другой сущности, то первая сущность признаётся владельцем связи, вторая сущность - владеемой. В однонаправленных отношениях сторона, которая имеет поле с типом другой сущности, является владельцем этой связи по умолчанию.	
Что такое каскады?	**Каскадирование - это когда мы выполняем какое-то действие с целевой Entity, то же самое действие будет применено к связанной Entity.** JPA CascadeType: ALL - гарантируют, что все персистентные события, которые происходят на родительском объекте, будут переданы дочернему объекту. PERSIST - означает, что операции save () или persist () каскадно передаются связанным объектам. (совершенно новый объекты добавляются) MERGE - означает, что связанные entity объединяются, когда объединяется entity-владелец. (сущно сть юудет заменена новой измененной сущностью)	
	REMOVE - удаляет все entity, связанные с удаляемой entity. DETACH - отключает все связанные entity, если происходит «ручное отключение». REFRESH - повторно считывают значение данного экземпляра и связанных сущностей из базы данных при вызове refresh().	После запуска метода flushAndClear() - обновления объекта Customer отправятся в базу данных, и произойдет следующее: 1. Hibernate заметит, что у объекта Customer уже не 4, а 3 связанных дочерних объекта Order; 2. в связи с этим Hibernate найдёт в таблице Order строку с удаленным объектом из коллекции Order; 3. очистит в этой строке ячейку с внешним ключом на Customer; 4. после чего удалит саму эту строку, как осиротевшую (более не ссылающуюся на родителя). Если не будет атрибута orphanRemoval = true, то пункт 4 не выполнится, и в таблице Order останется сущность Order, не связанная ни с одной сущностью Customer, то есть её ячейка с внешним ключом будет пустой. Такая сущность будет считаться осиротевшей.
Разница между PERSIST и MERGE?	**persist**(entity) следует использовать с совершенно новыми объектами, чтобы добавить их в БД (Если же объект уже есть в базе, то ничего не произойдет. Однако при попытки сохранения в базу объект со статусом Detached исключение EntityExistsException). Но в случае **merge**(entity) сущность, которая уже управляется в контексте персистентности, будет заменена новой сущностью (обновленной), и копия этой обновленной сущности вернется обратно. Но рекомендуется использовать для уже сохраненных сущностей.	**orphanRemoval. Директива orphanRemoval объявляет, что связанные экземпляры сущностей должны быть удалены, когда они отсоединены от родителя, или эквивалентно, когда родитель удален**
Какие два типа fetch стратегии в JPA вы знаете?	1) **LAZY** — **Hibernate может загружать данные не сразу, а при первом обращении к ним**, но так как это необязательное требование, то Hibernate имеет право изменить это поведение и загружать их сразу. Это поведение по умолчанию для полей, аннотированных @OneToMany, @ManyToMany и @ElementCollection. **В объект загружается прокси lazy-поля.** 2) **EAGER** — **данные поля будут загруженны немедленно**. Это поведение по умолчанию для полей, аннотированных @Basic, @ManyToOne и @OneToOne.	
Какие четыре статуса жизненного цикла Entity объекта (Entity Instance's Life Cycle) вы можете перечислить?	**Transient (New) —** свежесозданная оператором new() сущность не имеет связи с базой данных, не имеет данных в базе данных и не имеет сгенерированных первичных ключей. **managed** - объект создан, сохранён в бд, имеет primary key, управляется JPA **detached** -объект создан, имеет primary key, не является (или больше не является) частью контекста персистентности (не управляется JPA); **removed** - объект создан, управляется JPA, будет удален при commit-е и статус станет опять detached	

Как влияет операция persist на Entity объекты каждого из четырех статусов?	**new** → managed, и объект будет сохранен в базу при commit-е транзакции или в результате flush операций **managed** → операция игнорируется, однако зависимые Entity могут поменять статус на managed, если у них есть аннотации каскадных изменений **detached** → exception сразу или на этапе commit-а транзакции **removed** → managed, но только в рамках одной транзакции.
Как влияет операция remove на Entity объекты каждого из четырех статусов?	**new** → операция игнорируется, однако зависимые Entity могут поменять статус на removed, если у них есть аннотации каскадных изменений и они имели статус managed **managed** → removed и запись объект в базе данных будет удалена при commit-е транзакции (также произойдут операции remove для всех каскадно зависимых объектов) **detached** → exception сразу или на этапе commit-а транзакции **removed** → операция игнорируется
Как влияет операция merge на Entity объекты каждого из четырех статусов?	**new** → будет создан новый managed entity, в который будут скопированы данные прошлого объекта **managed** → операция игнорируется, однако операция merge сработает на каскадно зависимые Entity, если их статус не managed **detached** → либо данные будут скопированы в существующий managed entity с тем же первичным ключом, либо создан новый managed в который скопируются данные **removed** → exception сразу или на этапе commit-а транзакции
Как влияет операция refresh на Entity объекты каждого из четырех статусов?	**managed** → будут восстановлены все изменения из базы данных данного Entity, также произойдет refresh всех каскадно зависимых объектов **new**, **removed**, **detached** → exception
Как влияет операция detach на Entity объекты каждого из четырех статусов?	**managed**, **removed** → detached. **new**, **detached** → операция игнорируется
Для чего нужна аннотация Basic?	**@Basic - указывает на простейший тип маппинга данных на колонку таблицы базы данных.** Также в параметрах аннотации можно указать fetch стратегию доступа к полю и является ли это поле обязательным или нет. Может быть применена к полю любого из следующих типов: 1. Примитивы и их обертки. 2. java.lang.String 3. java.math.BigInteger 4. java.math.BigDecimal 5. java.util.Date 6. java.util.Calendar 7. java.sql.Date 8. java.sql.Time 9. java.sql.Timestamp 10. byte[] or Byte[] 11. char[] or Character[] 12. enums 13. любые другие типы, которые реализуют Serializable. аннотация @Basic означает, что это базовый тип, и Hibernate должен использовать стандартное сопоставление для его сохранения. Вообще, аннотацию @Basic можно не ставить, как это и происходит по умолчанию.
	Аннотация @Basic определяет 2 атрибута: 1. optional - boolean (по умолчанию true) - определяет, может ли значение поля или свойства быть null. Игнорируется для примитивных типов. Но если тип поля не примитивного типа, то при попытке сохранения сущности будет выброшено исключение. 2. fetch - FetchType (по умолчанию EAGER) - определяет, должен ли этот атрибут извлекаться незамедлительно (EAGER) или лениво (LAZY). Однако, это необязательное требование JPA, и провайдерам разрешено незамедлительно загружать данные, даже для которых установлена ленивая загрузка. Без аннотации @Basic при получении сущности из БД по умолчанию её поля базового типа загружаются принудительно (EAGER) и значения этих полей могут быть null Ленивая загрузка будет иметь смысл только тогда, когда у нас есть большой Serializable объект, отображаемый как базовый тип, так как в этом случае стоимость доступа к полю может быть значительной.

	Итак, если нам нужна отложенная загрузка атрибута и указание имени столбца, мы можем сказать
	
	`@Basic(fetch=FetchType.LAZY)`
	`@Column(name="WIBBLE")`
	
	Если бы нам нужно было поведение по умолчанию, не ленивое, то этого `@Column` было бы достаточно.

	@Column сопоставляет поле класса столбцу таблицы, а её атрибуты определяют поведение в этом столбце, используется для генерации схемы базы данных
Для чего нужна аннотация Column?	@Basic vs @Column: 1. Атрибуты @Basic применяются к сущностям JPA, тогда как атрибуты @Column применяются к столбцам базы данных. 2. @Basic имеет атрибут optional, который говорит о том, может ли поле объекта быть null или нет; с другой стороны атрибут nullable аннотации @Column указывает, может ли соответствующий столбец в таблице быть null. 3. Мы можем использовать @Basic, чтобы указать, что поле должно быть загружено лениво. 4. Аннотация @Column позволяет нам указать имя столбца в таблице и ряд других свойств: a. insertable/updatable - можно ли добавлять/изменять данные в колонке, по умолчанию true; b. length - длина, для строковых типов данных, по умолчанию 255. Коротко, в Column (колум) мы задаем constraints (констрейнтс), а в Basic (бейсик) - ФЕТЧ ТАЙП
	@Column(name="STUDENT_NAME", length=50, nullable=false, unique=false) private String name;

Для чего нужна аннотация Access?	***Она определяет тип доступа (access type) для класса entity, Mapped Superclass, embeddable или отдельных атрибутов, то есть как JPA будет обращаться к атрибутам entity, как к полям класса (FIELD) или как к свойствам класса (PROPERTY), имеющие геттеры (getter) и сетеры (setter).*** Определяет тип доступа к полям сущности. Для чтения и записи этих полей есть два подхода: 1. Field access (доступ по полям). При таком способе аннотации маппинга (Id, Column,...) размещаются над полями, и Hibernate напрямую работает с полями сущности, читая и записывая их. 2. Property access (доступ по свойствам). При таком способе аннотации размещаются над методами-геттерами, но никак не над сеттерами. Hibernate использует их и сеттеры для чтения и записи полей сущности.	**Но есть требование - у сущности с property access** названия методов должны соответствовать требованиям JavaBeans. Например, если у сущности Customer есть поле с именем firstName, то у этой сущности должны быть определены методы getFirstName и setFirstName для чтения и записи поля firstName.
	По умолчанию тип доступа определяется местом, в котором находится аннотация @Id. Если она будет над полем - это будет AccessType.FIELD, если над геттером - это AccessType.PROPERTY. Чтобы явно определить тип доступа у сущности, нужно использовать аннотацию @Access, которая может быть указана у сущности, Mapped Superclass и Embeddable class, а также над полями или методами. Поля, унаследованные от суперкласса, имеют тип доступа этого суперкласса. Когда у одной сущности определены разные типы доступа, то нужно использовать аннотацию @Transient для избежания дублирования маппинга.	
Для чего нужна аннотация @Cacheable?	**@Cacheable - необязательная аннотация JPA, используется для указания того, должна ли сущность храниться в кэше второго уровня.** JPA говорит о пяти значениях shared-cache-mode из persistence.xml, который определяет как будет использоваться second-level cache: ❖ **ENABLE_SELECTIVE**: (дефолтное и рекомендуемое значение):только сущности с аннотацией @Cacheable (равносильно значению по умолчанию @Cacheable(value=true)) будут сохраняться в кэше второго уровня. ❖ **DISABLE_SELECTIVE**: все сущности будут сохраняться в кэше второго уровня, за исключением сущностей, помеченных @Cacheable(value=false) как некэшируемые. ❖ **ALL**: сущности всегда кэшируются, даже если они помечены как некэшируемые. ❖ **NONE**: ни одна сущность не кэшируется, даже если помечена как кэшируемая. При данной опции имеет смысл вообще отключить кэш второго уровня. ❖ **UNSPECIFIED**: применяются значения по умолчанию для кэша второго уровня, определенные Hibernate. Это эквивалентно тому, что вообще не используется shared-cache-mode, так как Hibernate не включает кэш второго уровня, если используется режим UNSPECIFIED. Аннотация @Cacheable размещается над классом сущности. Её действие распространяется на эту сущность и её наследников, если они не определили другое поведение.	Как и большинство других полностью оснащенных платформ ORM, **Hibernate имеет концепцию кэша первого уровня.** Это кэш с областью действия сеанса, который гарантирует, что каждый экземпляр сущности загружается только один раз в постоянном контексте. **С другой стороны, кэш второго уровня имеет область SessionFactory** -, что означает, что он является общим для всех сеансов, созданных с помощью одной и той же фабрики сеансов.
Для чего нужна аннотация @Cache?	**Это аннотация Hibernate, настраивающая тонкости кэширования объекта в кэше второго уровня Hibernate.** @Cache принимает три параметра: ❖ **include** - имеет по умолчанию значение **all** и означающий кэширование всего объекта. Второе возможное значение - **non-lazy**, запрещает кэширование лениво загружаемых объектов. Кэш первого уровня не обращает внимания на эту директиву и всегда кэширует лениво загружаемые объекты. ❖ **region** - позволяет задать имя региона кэша для хранения сущности. Регион можно представить как разные области кэша, имеющие разные настройки на уровне реализации кэша. Например, можно было бы создать в конфигурации ehcache два региона, один с краткосрочным хранением объектов, другой с долгосрочным и отправлять часто изменяющиеся объекты в первый регион, а все остальные - во второй. Ehcache по умолчанию создает регион для каждой сущности с именем класса этой сущности, соответственно в этом регионе хранятся только эти сущности. К примеру, экземпляры Foo хранятся в Ehcache в кэше с именем "com.baeldung.hibernate.cache.model.Foo". ❖ **usage** - задаёт стратегию одновременного доступа к объектам. transactional read-write nonstrict-read-write read-only	
Для чего нужны аннотации @Embedded и @Embeddable?	@Embeddable - аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы. @Embedded - аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что мы внедряем встраиваемый класс.	Для отображения композиции в общую таблицу существуют аннотации @Embedable и @Embeded. Первая ставится над полем, а вторая над классом. Они взаимозаменяемые. (Композиция, когда не может существовать отдельно)
Как смапить составной ключ?	**Составной первичный ключ**, также называемый составным ключом, представляет собой **комбинацию из двух или более столбцов** для формирования первичного ключа таблицы.	
	@IdClass Допустим, у нас есть таблица с именем Account, и она имеет два столбца - accountNumber и accountType, которые формируют составной ключ. Чтобы обозначить оба этих поля как части составного ключа мы должны создать класс, например, ComplexKey с этими полями. Затем нам нужно аннотировать сущность Account аннотацией @IdClass(ComplexKey.class). Мы также должны объявить поля из класса ComplexKey в сущности Account с такими же именами и аннотировать их с помощью @Id.	
	@EmbeddedId Рассмотрим пример, в котором мы должны сохранить некоторую информацию о книге с заголовком и языком в качестве полей первичного ключа. В этом случае класс первичного ключа, BookId, должен быть аннотирован @Embeddable. Затем нам нужно встроить этот класс в сущность Book, используя @EmbeddedId.	
Для чего нужна аннотация ID? Какие @GeneratedValue вы знаете?	**Аннотация @Id определяет простой (не составной) первичный ключ**, состоящий из одного поля. В соответствии с JPA, допустимые типы атрибутов для первичного ключа: 1. примитивные типы и их обертки; 2. строки; 3. BigDecimal и BigInteger; 4. java.util.Date и java.sql.Date.	
	Если мы хотим, чтобы значение первичного ключа генерировалось для нас автоматически, мы можем добавить первичному ключу, отмеченному аннотацией @Id, аннотацию **@GeneratedValue**. Возможно 4 варианта: **AUTO(default) -** Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию. Поведение по умолчанию - исходить из типа поля идентификатора. **IDENTITY -** для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются вне текущей выполняемой транзакции(на стороне базы, так что этого столбца мы не увидим), что позволяет базе данных генерировать новое значение при каждой операции вставки. В промежутках транзакций сущность будет сохранена.	

SEQUENCE - тип генерации, рекомендуемый документацией Hibernate. Для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). В бд можно будет увидеть дополнительную таблицу. Но если наша БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE. В промежутках транзакций сущность не будет сохранена, так как хибер возьмет из таблицы id hibernate-sequence и вернётся обратно в приложение.
SEQUENCE - это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе.
TABLE - Hibernate должен получать первичные ключи для сущностей из создаваемой для этих целей таблицы, способной содержать именованные сегменты значений для любого количества сущностей. Требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок и замедляет работу приложения.

```
@TableGenerator(name = "table-generator",
    table = "dep_ids",
    pkColumnName = "seq_id",
    valueColumnName = "seq_value")
private long depId;

// ...
}
```

dClass

Расскажите про аннотации @JoinColumn и @JoinTable? Где и для чего они используются ?

@JoinColumn используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями или коллекциями.
Мы помним, что только сущность-владелец связи может иметь внешние ключи от другой сущности (владеемой). Однако, мы можем указать @JoinColumn как во владеющей таблице, так и во владеемой, но столбец с внешними ключами всё равно появится во владеющей таблице.
Особенности использования:

- @OneToOne: означает, что появится столбец в таблице сущности-владельца связи, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности.

- @OneToMany/@ManyToOne: если не указать на владеемой стороне связи атрибут mappedBy, создается joinTable с ключами обеих таблиц. Но при этом же у владельца создается столбец с внешними ключами.

- @OneToOne означает, что появится столбец addressId в таблице сущности-владельца связи Office, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности Address.

```
@Entity
public class Office {
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "addressId")
    private Address address;
}
```

- @OneToMany/@ManyToOne: в данном случае мы можем использовать параметр mappedBy для того, чтобы столбец с внешними ключами находился на владеющей стороне ManyToOne - то есть в таблице Email:

```
@Entity
public class Employee {

    @Id
    private Long id;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
    private List<Email> emails;
}

@Entity
public class Email {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

В приведенном выше примере таблица Email (владелец связи) имеет столбец employee_id, в котором хранится значение идентификатора и внешний ключ к таблице Employee.

@JoinColumns используется для группировки нескольких аннотаций @JoinColumn, которые используются при установлении связей между сущностями или коллекциями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа.
В каждой аннотации @JoinColumn должны быть указаны элементы name и referencedColumnName.

@JoinTable используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами. *@JoinTable*
(name = "CATALOG", joinColumns = @JoinColumn(name = "ID_BOOK"), inverseJoinColumns = @JoinColumn(name = "ID_STUDENT"))
private Student student;

Для чего нужны аннотации @OrderBy и @OrderColumn , чем они отличаются?

@OrderBy указывает порядок, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов **при их извлечении из БД. Если в кэше есть нужные данные, то сортировки не будет.** Так как @OrderBy просто добавляет к sql-запросу Order By, а при получении данных из кэша, обращения к бд нет. Эта аннотация может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany.
При использовании с коллекциями базовых типов, которые имеют аннотацию **@ElementCollection, элементы этой коллекции будут отсортированы в натуральном порядке, по значению базовых типов.**

Если это коллекция встраиваемых типов (@Embeddable), то используя точку (".") мы можем сослаться на атрибут внутри встроенного атрибута. Например, следующий код отсортирует адреса по названиям стран в обратном порядке:

```
@Embeddable
public class Address {
    ...
    @Embedded
    private City city
    ....
}
@ElementCollection
@OrderBy("city.country DESC")
private List<Address> addresses;
```

Если это коллекция **сущностей**, то у аннотации @OrderBy можно указать имя поля сущности, по которому сортировать эти самые сущности.

```
@Entity
public class Task {
    ...
    @OneToOne
    private Employee supervisor;
    ...
}
@ManyToMany
@OrderBy("supervisor")
private List<Task> tasks;
```

Если это коллекция встраиваемых типов (@Embeddable), то используя точку (".") мы можем сослаться на атрибут внутри встроенного атрибута.
Если это коллекция сущностей, то у аннотации @OrderBy можно указать имя поля сущности, по которому сортировать эти самые сущности:
Если мы не укажем у @OrderBy параметр, то сущности будут упорядочены по первичному ключу.
В случае с сущностями доступ к полю по точке не работает. Попытка использовать вложенное свойство, например @OrderBy ("supervisor.name") повлечет Runtime Exceprtion.

@OrderColumn создает в таблице столбец с индексами порядка элементов, который используется для поддержания постоянного порядка в списке, но этот столбец не считается частью состояния сущности или встраиваемого класса.
Hibernate отвечает за поддержание порядка как в базе данных при помощи столбца, так и при получении сущностей и элементов из БД. Hibernate отвечает за обновление порядка при записи в базу данных, чтобы отразить любое добавление, удаление или иное изменение порядка, влияющее на список в таблице.

```
'Show Columns from Employee_PhoneNumbers'
[EMPLOYEE_ID, BIGINT(19), NO, PRI, NULL]
[PHONENUMBERS, VARCHAR(255), YES, , NULL]
[PHONENUMBERS_ORDER, INTEGER(10), NO, PRI, NULL]
'Select * FROM Employee_PhoneNumbers'
[1, 111-111,111, 0]
[1, 222-222,222, 1]
[2, 333-333-333, 0]
[2, 444-444-444, 1]
[2, 666-666-666, 2]
[3, 555-555-555, 0]
```

@OrderBy vs @OrderColumn
Порядок, указанный в @OrderBy, применяется только в рантайме при выполнении запроса к БД, То есть в контексте персистентности, в то время как при использовании @OrderColumn, порядок сохраняется в отдельном столбце таблицы и поддерживается при каждой вставке/обновлении/удалении элементов.

Для чего нужна аннотация Transient?

@Transient используется для объявления того, какие поля у сущности, встраиваемого класса или Mapped SuperClass **НЕ БУДУТ сохранены в базе данных.**
Persistent fields (постоянные поля) - это поля, значения которых будут по умолчанию сохранены в БД. Ими являются любые не static и не final поля.
Transient fields (временные поля):

- static и final поля сущностей;
- иные поля, объявленные явно с использованием Java-модификатора transient, либо JPA-аннотации @Transient.

```
Initial user: User{firstName='Stefan', lastName='Smith',
email='ssmith@email.com', birthDate=1991-07-16, login='ssmith',
password='gemma_arterton_4ever_in_my_heart91'} Loaded user from file:
User{firstName='Stefan', lastName='Smith', email='ssmith@email.com',
birthDate=1991-07-16, login='ssmith', password='null'}
```

ы можем получить

етод доступа к полю в

ие о полях, которые

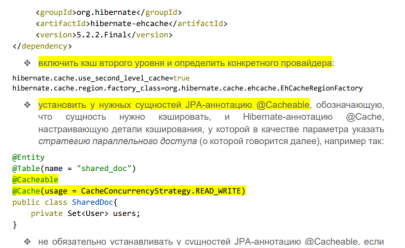
грегированы в классе,

Какие шесть видов блокировок (lock) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)?	В порядке от самого ненадежного и быстрого, до самого надежного и медленного: 1. **NONE** — без блокировки. 2. **OPTIMISTIC** (синоним READ в JPA 1) — оптимистическая блокировка, которая работает, так: если при завершении транзакции кто-то извне изменит поле @Version, то будет сделан RollBack транзакции и будет выброшено OptimisticLockException. 3. **OPTIMISTIC_FORCE_INCREMENT** (синоним WRITE в JPA 1) — работает по тому же алгоритму, что и LockModeType.OPTIMISTIC за тем исключением, что после commit значение поле Version принудительно увеличивается на 1. В итоге окончательно после каждого коммита поле увеличится на 2(увеличение, которое можно увидеть в Post-Update + принудительное увеличение). 4. **PESSIMISTIC_READ** — данные блокируются в момент чтения и это гарантирует, что никто в ходе выполнения транзакции не сможет их изменить. Остальные транзакции, тем не менее, смогут параллельно читать эти данные. Использование этой блокировки может вызывать долгое ожидание блокировки или даже выкидывание PessimisticLockException. 5. **PESSIMISTIC_WRITE** — данные блокируются в момент записи и никто с момента захвата блокировки **не может в них писать и не может их читать** до окончания транзакции, владеющей блокировкой. Использование этой блокировки может вызывать долгое ожидание блокировки. 6. **PESSIMISTIC_FORCE_INCREMENT** — ведёт себя как PESSIMISTIC_WRITE, но в конце транзакции увеличивает значение поля @Version, даже если фактически сущность не изменилась.	

	Оптимистичное блокирование - подход предполагает, **что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им свободно выполнять любые чтения и обновления данных.** Но при окончании транзакции производится проверка, изменились ли данные в ходе выполнения данной транзакции и, если да, транзакция обрывается и выбрасывается OptimisticLockException. Оптимистичное блокирование в JPA реализовано путём внедрения в сущность специального поля версии:
	@Version
	private long version;
	Поле, аннотирование @Version, может быть целочисленным или временнЫм. При завершении транзакции, если сущность была заблокирована оптимистично, будет проверено, не изменилось ли значение @Version кем-либо ещё, после того как данные были прочитаны, и, если изменилось, будет выкинуто OptimisticLockException. Использование этого поля позволяет отказаться от блокировок на уровне базы данных и сделать всё на уровне JPA, улучшая уровень конкурентности.
	Позволяет отказаться от блокировок на уровне БД и делать всё с JPA.
		Обе блокировки ставятся путём вызова метода lock() у EntityManager, в который передаётся сущность, требующая блокировки и уровень блокировки:
		`EntityManager em = entityManagerFactory.createEntityManager();`
		`em.lock(company1, LockModeType.OPTIMISTIC);`
		`em.lock(company2, LockModeType.OPTIMISTIC_FORCE_INCREMENT);`
		Блокировка будет автоматически снята при завершении транзакции, снять её до этого вручную невозможно.

	Пессимистичное блокирование - подход напротив, ориентирован на транзакции, которые часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным в тот момент когда читает их. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение). **Пессимистичное блокирование выполняется на уровне базы и поэтому не требует вмешательств в код сущности.**	
	Блокировки ставятся путём вызова метода lock() у EntityManager, в который передаётся сущность, требующая блокировки и уровень блокировки: EntityManager em = entityManagerFactory.createEntityManager(); em.lock(company1, LockModeType.OPTIMISTIC);	
Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?	1. first-level cache (кэш первого уровня) — кэширует данные одной транзакции; 2. second-level cache (кэш второго уровня) — кэширует данные транзакций от одной фабрики сессий. Провайдер JPA может, но не обязан реализовывать работу с кэшем второго уровня. 3. Кэш запросов.	
	Кэш первого уровня – это кэш сессии (Session), который является обязательным, это и есть PersistenceContext. Через него проходят все запросы. В том случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление этого объекта для того, чтобы сократить количество выполненных запросов в БД. Например, при пяти истребованиях одного и того же объекта из БД в рамках одного persistence context, запрос в БД будет выполнен один раз, а остальные четыре загрузки будут выполнены из кэша. Если мы закроем сессию, то все объекты, находящиеся в кэше, теряются, а далее – либо сохраняются в БД, либо обновляются.	

	Особенности кэша первого уровня:
	❖ включен по умолчанию, его нельзя отключить;
	❖ связан с сессией (контекстом персистентности), то есть разные сессии видят только объекты из своего кэша, и не видят объекты, находящиеся в кэшах других сессий;
	❖ при закрытии сессии PersistenceContext очищается - кэшированные объекты, находившиеся в нем, удаляются;
	❖ при первом запросе сущности из БД, она загружается в кэш, связанный с этой сессией;
	❖ если в рамках этой же сессии мы снова запросим эту же сущность из БД, то она будет загружена из кэша, и никакого второго SQL-запроса в БД сделано не будет;
	❖ сущность можно удалить из кэша сессии методом **evict(),** после чего следующая попытка получить эту же сущность повлечет обращение к базе данных;
	❖ метод **clear()** очищает весь кэш сессии.
		load() или объекта с лениво загружаемыми полями, лениво загружаемые данные в кэш не попадут, а вместо этих данных Hibernate создаст объект Proxy. Однако, как только мы обратимся к этому прокси-объекту в рамках этого же открытого контекста персистентности, Hibernate всё-таки выполнит запрос в базу и данные будут загружены и в объект и в кэш. А вот следующая попытка лениво загрузить объект приведёт к тому, что объект сразу вернут из кэша уже полностью загруженным, без обращения в БД.
		Кэш запросов (Query Cache)
		Результаты HQL-запросов также могут быть кэшированы. Это полезно, если мы часто выполняем запрос к объектам, которые редко меняются. Чтобы включить кэш запросов, установите для свойства hibernate.cache.use_query_cache значение true:
		`hibernate.cache.use_query_cache=true`
		Затем для **каждого** запроса мы должны явно указать, что запрос кэшируется через подсказку в запросе setHint("org.hibernate.cacheable", true):
		`entityManager.createQuery("select f from Foo f")`
		`.setHint("org.hibernate.cacheable", true)`
		`.getResultList();`

	Если кэш первого уровня привязан к объекту сессии, то **кэш второго уровня** привязан к объекту-фабрике сессий (Session Factory object) и, следовательно, кэш второго уровня доступен одновременно в нескольких сессиях или контекстах персистентности. Кэш второго уровня требует некоторой настройки и поэтому не включен по умолчанию. Настройка кэша заключается в конфигурировании реализации кэша и разрешения сущностям быть закэшированными. Hibernate не реализует сам никакого in-memory cache, а использует существующие реализации кэшей.	
Как работать с кешем 2 уровня?	**Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня.** Hibernate поставляется со встроенной поддержкой стандарта кэширования Java JCache, а также двух популярных библиотек кэширования: Ehcache и Infinispan.	
	В Hibernate кэширование второго уровня реализовано в виде абстракции, то есть мы должны предоставить любую её реализацию, вот несколько провайдеров: Ehcache, OSCache, SwarmCache, JBoss TreeCache. Для Hibernate требуется только реализация интерфейса org.hibernate.cache.spi.RegionFactory, который инкапсулирует все детали, относящиеся к конкретным провайдерам. По сути, RegionFactory действует как мост между Hibernate и поставщиками кэша. В примерах будем использовать Ehcache. **Что нужно сделать:** ❖ добавить мавен-зависимость кэш-провайдера нужной версии ❖ включить кэш второго уровня и определить конкретного провайдера hibernate.cache.use_second_level_cache=true hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory ❖ установить у нужных сущностей JPA-аннотацию @Cacheable, обозначающую, что сущность нужно кэшировать, и Hibernate-аннотацию @Cache, настраивающую детали кэширования, у которой в качестве параметра указать стратегию параллельного доступа" на "Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня. Hibernate поставляется со встроенной поддержкой стандарта кэширования Java JCache, а также двух популярных библиотек кэширования: Ehcache и Infinispan. **Shared Cache Mode** ENABLE_SELECTIVE, ALL, NONE, UNSPECIFIED	 ❖ включить кэш второго уровня и определить конкретного провайдера: ❖ установить у нужных сущностей JPA-аннотацию @Cacheable, обозначающую, что сущность нужно кэшировать, и Hibernate-аннотацию @Cache, настраивающую детали кэширования, у которой в качестве параметра указать *стратегию параллельного доступа* (о которой говорится далее), например так: ❖ не обязательно устанавливать у сущностей JPA-аннотацию @Cacheable. если

уровня?

Стратегии параллельного доступа к объектам Проблема заключается в том, что кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом. Следовательно надо как-то обеспечивать их одинаковым представлением этого объекта.

- **READ_ONLY: Используется только для сущностей, которые никогда не изменяются** (будет выброшено исключение, если попытаться обновить такую сущность). Очень просто и производительно. Подходит для некоторых статических данных, которые не меняются.
- **NONSTRICT_READ_WRITE**: **Кэш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их.** Таким образом, строгая согласованность не гарантируется, и существует небольшое временное окно между обновлением данных в БД и обновлением тех же данных в кэше, во время которого параллельная транзакция может получить из кэша устаревшие данные.
- **READ_WRITE**: Эта стратегия гарантирует строгую согласованность, которую она достигает, используя «мягкие» блокировки: **когда обновляется кэшированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции.** Все параллельные транзакции, которые пытаются получить доступ к записям в кэше с наложенной мягкой блокировкой, не смогут их прочитать или записать и отправят запрос в БД. **Ehcache использует эту стратегию по умолчанию.**
- **TRANSACTIONAL: полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, словно они работали с ними последовательно одна транзакция за другой. Плата за это — блокировки и потеря производительности**

- Не обязательно устанавливать у сущности JPA-аннотацию @Cacheable, если работаем с Hibernate напрямую, не через JPA.
- чтобы кэш не "съел" всю доступную память, можно, например, ограничивать количество каждого типа сущностей, хранимых в кэше:

```
<ehcache>
    <cache name="com.baeldung.persistence.model.Foo" maxElementsInMemory="1000"/>
</ehcache>
```

Hibernate Proxy используется для замены реальной сущности POJO (Plain Old Java Object).

Что такое JPQL/HQL и чем он отличается от SQL?

Hibernate Query Language (**HQL**) и Java Persistence Query Language (**JPQL**) - оба являются объектно-ориентированными языками запросов, схожими по природе с SQL. JPQL - это подмножество HQL.

JPQL - это язык запросов, практически такой же как SQL, однако, вместо имен и колонок таблиц базы данных, он использует имена классов Entity и их атрибуты. В качестве параметров запросов также используются типы данных атрибутов Entity, а не полей баз данных. В отличии от SQL в JPQL есть автоматический полиморфизм, то есть **каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков**, независимо от стратегии наследования. В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager.

Полиморфные запросы

В отличии от SQL в запросах JPQL есть автоматический полиморфизм, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков, независимо от стратегии наследования (например, запрос select * from Animal, вернет не только объекты Animal, но и объекты классов Cat и Dog, которые унаследованы от Animal). Чтобы исключить такое поведение используется функция TYPE в where условии (например select * from Animal a where TYPE(a) IN (Animal, Cat) уже не вернет объекты класса Dog).

В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager. Для создания Query или TypedQuery необходимо использовать метод EntityManager#createQuery. Для именованных запросов необходим метод EntityManager#createNamedQuery.

Запросы, как обычные, так и именованные, формируются из EntityManager:

```
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name"
);
TypedQuery<Person> typedQuery = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name", Person.class
);
@NamedQuery(
    name = "get_person_by_name",
    query = "select p from Person p where name = :name"
)
```

В Hibernate HQL-запрос представлен org.hibernate.query.Query, полученный из Session. Если HQL является именованным запросом, то будет использоваться Session#getNamedQuery, в противном случае требуется Session#createQuery.

Что такое Criteria API и для чего он используется?

Начиная с версии 5.2 Hibernate Criteria API объявлен deprecated. Вместо него рекомендуется использовать JPA Criteria API.
JPA Criteria API - это актуальный API, используемый только для выборки(select) сущностей из БД в более объектно-ориентированном стиле.
Основные преимущества JPA Criteria API:

- ошибки могут быть обнаружены во время компиляции;
- позволяет динамически формировать запросы на этапе выполнения приложения.

Основные недостатки:

- нет контроля над запросом, сложно отловить ошибку
- влияет на производительность, множество классов

Для динамических запросов - фрагменты кода создаются во время выполнения - JPA Criteria API является предпочтительней.

API нельзя. Критерии используются **только для выборки из базы данных** в более объектно-ориентированном стиле. Используется для динамических запросов. Запросы выглядят так:

```
session.createCriteria(Person.class)
  .setMaxResults(10)
  .list()
  .forEach(System.out::println);
```

Запрос выше полностью аналогичен запросу HQL "from Person". С Criteria также работают и все те вещи, которые работают и с Query: пейджинг, таймауты и т.д.

Разумеется, в Criteria запросах можно и нужно накладывать условия, по которым объекты будут отбираться:

```
session.createCriteria(Person.class)
  .add(Restrictions.eq("lastName", "Testoff"))
  .list()
  .forEach(System.out::println);
```

Вот некоторые области применения Criteria API:
Criteria API поддерживает проекцию, которую мы можем использовать для агрегатных функций вроде sum(), min(), max() и т.д.
Criteria API может использовать ProjectionList для извлечения данных только из выбранных колонок.
Criteria API может быть использована для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection().
Criteria API поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add() с помощью которого добавляются ограничения (Restrictions).
Criteria API позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().

Проблема N+1 запросов возникает, **когда получение данных из БД выполняется за N дополнительных SQL-запросов для извлечения тех же данных, которые могли быть получены при выполнении основного SQL-запроса**.

1. **JOIN FETCH**

И при FetchType.EAGER и при FetchType.LAZY нам поможет JPQL-запрос с JOIN FETCH. Опцию «FETCH» можно использовать в JOIN (INNER JOIN или LEFT JOIN) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта.
Лучший вариант решения для простых запросов (1-3 уровня вложенности связанных объектов).

```
select pc
   from PostComment pc
   join fetch pc.post p
```

2. **EntityGraph**

В случаях, когда нам нужно получить по-настоящему много данных, и у нас jpql запрос - лучше всего использовать EntityGraph.

Допустим у нас есть две сущности Post и PostComment. В БД имеется 4 Post и у каждого из них по одному PostComment:

```
@Entity(name = "Post")
@Table(name = "post")
public class Post {
    @Id
    private Long id;
    private String title;
    //Getters and setters omitted for brevity
}

@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {
    @Id
    private Long id;
    @ManyToOne
    private Post post;
    private String review;
    //Getters and setters omitted for brevity
}
```

N+1 при FetchType.EAGER

Так как у @ManyToOne план извлечения по умолчанию - EAGER, то при получении из БД сущности PostComment немедленно будет загружена связанная с ней

N+1 при FetchType.LAZY

Даже если мы явно переключимся на использование FetchType.LAZY для всех ассоциаций, мы всё равно можем столкнуться с проблемой N+1. Явно укажем над полем Post план извлечения - LAZY:

```
@ManyToOne(fetch = FetchType.LAZY)
private Post post;
```

Теперь, когда мы выбираем все сущности PostComment, Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc
```

Но, если в этом же контексте персистентности, мы обратимся к сущностям Post в PostComment:

```
for(PostComment comment : comments) {
    LOGGER.info(
```

Расскажите про проблему N+1 Select и путях ее решения.

3. **@Fetch(FetchMode.SUBSELECT)**

Аннотация Hibernate. **Можно использовать только с коллекциями.** Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
@Fetch(value = FetchMode.SUBSELECT)
    private Set<Order> orders = new HashSet<>();
```

Первый запрос:

```
select ...
from customer customer0_
```

Второй запрос:

```
select ...
    from
        order order0_
    where
        order0_.customer_id in (
            select
                customer0_.id
            from
                customer customer0_
        )
```

4. **Batch fetching**

Это Аннотация Hibernate, в JPA её нет. Указывается над классом сущности или над полем коллекции с ленивой загрузкой. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций. Количество загружаемых сущностей указывается в аннотации.

```
@BatchSize(size=5)
private Set<Order> orders = new HashSet<>();
```

каждой. Но так как у нас @BatchSize(size=5), то Hibernate сделает 3 запроса: в первом и втором получит по пять коллекций, а в третьем получит две коллекции.

5. **HibernateSpecificMapping, SqlResultSetMapping**

Для нативных запросов рекомендуется использовать именно их.

Что такое Entity Graph

Он позволяет определить шаблон путем группировки связанных полей, которые мы хотим получить, и позволяет нам выбирать тип графа во время выполнения.

Основная цель JPA Entity Graph - улучшить производительность в рантайме при загрузке базовых полей сущности и связанных сущностей и коллекций.

Вкратце, Hibernate загружает весь граф в одном SELECT-запросе, то есть все указанные связи от нужной нам сущности:

Entity Graphs — механизм динамического изменения fetchType для каждого запроса

получения из БД сущности PostComment немедленно будет загружена связанная с ней сущность Post:

```
List<PostComment> comments = entityManager.createQuery(
    """select pc from PostComment pc""", PostComment.class)
.getResultList();
```

Этот SQL-запрос приведет к проблеме N+1 и выполнит больше запросов, чем нужно:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc

SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=2
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=3
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=4
```

Обратите внимание на дополнительные 4 оператора SELECT, которые выполняются, потому что 4 сущности Post должны быть извлечены из БД до возврата списка из 4 сущностей PostComment с инициализированными полями Post. Автоматически Hibernate делает это не очень хорошо, порождая количество запросов в БД, равное N+1, а именно один запрос для получения PostComment, и четыре запроса для получения Post для каждого PostComment. В нашем случае это проблема 4+1.

```
        "The Post '{}' got this review '{}'",
        comment.getPost().getTitle(),
        comment.getReview()
    );
}
```

То мы опять получим проблему N+1:

```
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=1
-- The Post 'High-Performance Java Persistence - Part 1' got this review
-- 'Excellent book to understand Java Persistence'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=2
-- The Post 'High-Performance Java Persistence - Part 2' got this review
-- 'Must-read for Java developers'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=3
-- The Post 'High-Performance Java Persistence - Part 3' got this review
-- 'Five Stars'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=4
-- The Post 'High-Performance Java Persistence - Part 4' got this review
-- 'A great reference book'
```

)хотя использовать @BatchSize лучше, если столкнулся с проблемой N+1, в большинстве случаев гораздо лучшей альтернативой является использование DTO или JOIN FETCH, поскольку они позволяют получать все необходимые данные одним запросом.

В данном примере мы создали EntityGraph и при его помощи хотим, чтобы при загрузке сущности Post загружались поля subject, user, comments. Также мы захотели, чтобы у каждой сущности comments загружалось поле user, для чего мы использовали subgraphs.

EntityGraph можно определить и без помощи аннотаций, а используя entityManager из JPA API:

```
EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);
entityGraph.addAttributeNodes("subject");
entityGraph.addAttributeNodes("user");
entityGraph.addSubgraph("comments").addAttributeNodes("user");
```

JPA определяет два свойства или подсказки, с помощью которых Hibernate может выбирать стратегию извлечения графа сущностей во время выполнения:

- fetchgraph - из базы данных извлекаются только указанные в графе атрибуты. *Поскольку мы используем Hibernate, мы можем заметить, что в отличие от спецификаций JPA, атрибуты, статически настроенные как EAGER, также загружаются.*
- loadgraph - в дополнение к указанным в графе атрибутам, также извлекаются атрибуты, статически настроенные как EAGER.

В любом случае, первичный ключ и версия, если таковые имеются, всегда загружаются.